

FAQ

Q What's the difference between managed and unmanaged code?

A Unmanaged code is raw machine code executed by the machine's processor. This was the norm until recently, when Microsoft introduced the term 'managed code' to refer to code executed under the control of a virtual machine (VM).

A VM is software written to run on a real machine. Any software written for that VM can be run on any real computer running that VM without needing to be recoded. The VM takes care of turning it into a form the computer understands.

This is the way Java works: each real machine has a Java client that's capable of running Java programs written once.

Managed code is good for safety, as the VM can inspect the code and make sure it's not trying to do anything dangerous. But unmanaged code is faster.

Mike James

IT author, lecturer
and programmer

mike@computershopper.co.uk

Work with gadgets

It's easy to personalise Vista's sidebar with gadgets to improve access to frequently used tools. Mike James shows you how to use the Gadget API to get things organised



Windows Vista's sidebar introduced mini-applications called gadgets that are built using HTML, XML and script. Vista ships with a range of built-in gadgets, including Weather report, an RSS news reader and a PC system monitor. There are also hundreds of extras that can be downloaded easily. Gadgets are quite easy to write yourself. In this *Programming Expert* we show you how to create one.

A SIMPLE GADGET

A gadget is just an HTML page plus script run in the context of the sidebar. However, the sidebar provides a range of functions and other facilities that are not standard in a browser window. It is the Gadget API that really makes gadgets more than just a webpage.

The simplest gadget HTML file is:

```
<html>␣
<head>␣
  <title>My First Gadget</title>␣
</head>␣
<body>␣
  Gadget Starter␣
</body>␣
</html>␣
```

This HTML file can be created using Notepad or any text editor that can save a simple text file.

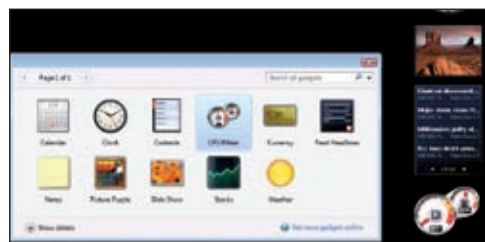
You need to create a new directory in Users\username\AppData\Local\Microsoft\Windows Sidebar\Gadgets. Gadgets stored in this directory are only

available to the named user. If you want your gadget to be available to all users you need to create the directory in Program Files\Windows Sidebar\Gadgets. If you can't see these directories, make sure you have used the Folders, Options menu to display hidden files.

The name of the directory that you create has to take the form name.gadget. For our example, create a directory, in the Gadget Directory of your choice, called MyGadget1.gadget and save into this directory the HTML listed above in a file called MyGadget1.HTML. All that we need to complete our first gadget is an XML manifest file that describes the gadget to the sidebar, so that it can load it and provide additional information to the user.

There are lots of optional XML tags that can be included in the manifest, but the simplest manifest for our gadget is:

```
<?xml version="1.0" encoding="utf-8" ?>␣
<gadget>␣
  <name>My First Gadget </name>␣
  <version>1.0.0.0</version>␣
  <hosts>␣
    <host name="sidebar">␣
      <base type="HTML" apiVersion="1.0.0"␣
        src="MyGadget1.html" />␣
      <permissions>Full</permissions>␣
      <platform minPlatformVersion="1.0" />␣
    </host>␣
  </hosts>␣
</gadget>␣
```



▲ Vista's built-in Gadgets

The name tag is displayed in the Gadget Gallery and on the Sidebar window. The version is used to check and resolve problems if different versions of the same gadget are installed. The Hosts section will, in future, let you specify a range of possible hosts for the gadget, but at the moment the only possibility is the sidebar. The base tag gives the type of the gadget, which is always HTML, the name of the source file containing the HTML. The manifest has to be stored in the same directory and is always called gadget.xml. With these two files stored in

the Gadget Directory you can click on the '+' button at the top of the sidebar and the Gadget Gallery appears, complete with My First Gadget, which is installed if you double-click it.

At this early stage, you can package a gadget in a Zip or CAB file. For example, if you place all of the gadget's files, not including the directory, into a Zip and change the extension to .gadget, your gadget will be installed into the Gadget Directory when the user double-clicks on it.

GOING FURTHER

As it stands, the gadget's display area is too small to show even its short message. You have to specify a display size using a Style attribute. Most of the examples of simple gadgets use a separate style or even a style sheet, but all you really need to do is to change the <body> tag to read:

```
<body style="width:130; height:50 " >
```

With the change to the <body> tag, the gadget's message is now fully visible. Most gadgets in the sidebar are about 130 pixels wide. If you make the width much larger the gadget spills over on to the desktop. Gadgets can be detached from the sidebar, where widths of more than 130 pixels are less of a problem. You can set any height you want, but you risk filling the sidebar with a tall gadget.

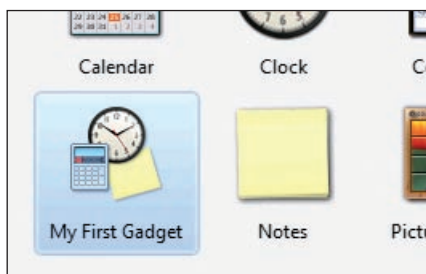
So far so good, but where next? To a degree, this is obvious if you know enough HTML. You can make the gadget look impressive simply by using what you already know about layout plus a few tips and tricks. For example, to create a non-rectangular gadget, simply use a PNG or GIF with a transparent border and set this to be the background image.

GADGET API

The Gadget API provides a whole collection of new objects that you can use to interact with the facilities of the sidebar and with the wider system. For example, all gadgets display a close box to the left of their display area. With the help of the API you can add a small spanner icon, which can be used to display a settings dialog box. The System.Gadget object provides methods and properties to control and monitor the gadget.

In this case we need the SettingsUI property, which specifies another HTML file that is loaded into the settings dialog box. For example, the simplest HTML file for a settings dialog box is:

```
<html>
<head>
</head>
<body style="width:250;
height:75">
  This is a settings dialog box
```



▲ My First Gadget ready for installation

```
</body>
</html>
```

Store this in a file called Settings.html. To make the Spanner icon appear, and to specify it as the HTML file to use, we are going to need some script. Change the header of the Gadget HTML to read:

```
<head>
<title>My First Gadget</title>
<script>
  System.Gadget.settingsUI=
    "settings.html";
</script>
</head>
```

When you install the gadget you will see a spanner icon on the right. When you click the spanner, your HTML file is displayed within a dialog box with OK and Cancel buttons.

We need to do a little more to make the settings box useful, but the new API provides properties that make persistent settings easy. The System.Gadget.Settings object provides a read and write method that will retrieve or store a named setting. To use this, we also need to handle the events generated when the user clicks the OK or Cancel button. In the Settings.html file we need to define two event handlers: one for System.Gadget.onSettingsClosing and one for window.onload:

```
<head>
<script>
window.onload=function()
{
  Text1.value=System.Gadget.
    Settings.read("Time");
}
```

The window.onload function simply reads the value of the Time setting and stores this in a text box. The first time the Settings box appears there is no value associated with Time, so the text box will be empty.

The onSettingsClosing event passes an event object that we can test to see which button has been clicked:

```
System.Gadget.onSettingsClosing =
  function(event)
{
  if (event.closeAction ==
    event.Action.commit)
  {
```

If the test equals event.Action.commit then the OK button was pressed and we need to save the current time in the Time setting:

```
    d=new Date();
    System.Gadget.Settings.write(
      "Time",d.getTime());
  }
  event.cancel = false;
}
</script>
```

We also need a text box so the HTML file ends:

```
</head>
<body style="width:250;
height:75">
  <input id="Text1" type="text" />
```



▲ A simple settings box

```
</body>
</html>
```

If you make these changes you will find that now when you first open the Settings dialog box the text box is empty. If you then click OK, the time is stored and the next time you open Settings this is what you see in the text box. Each time you click OK the time is updated, but if you click Cancel it is left unchanged. We used the JavaScript date object to retrieve the time.

An alternative is to use the Time objects provided by the API:

```
LocalTime = System.Time.
  getLocalTime(
    System.Time.currentTimeZone);
System.Gadget.Settings.write(
  "Time",LocalTime);
```

DEBUGGING

Unless you have a full copy of Visual Studio, debugging a gadget can be difficult because the alert function has been disabled and you can't open new windows. In addition to the settings window, a gadget can open another type of window called a flyout. The gadget can have as many flyout windows as it needs, but only one can be on display at a time.

To make debugging easier, and to provide an example of using the flyout, let's create a debug flyout. First we need an HTML page to be used as the basis for the flyout:

```
<html>
<head>
</head>
<body style="width: 130; height:
75">
  Debug Message
  <br />
  <div id="flyoutContent">
  </div>
  <input id="Button1"
    type="button"
    value="OK"
    onclick="System.Gadget.Flyout.
      show=false;" />
</body>
</html>
```

The <div> tag will be used to receive the text we want to display. The button is used to dismiss the flyout and sets its show property to false. The difference between the Settings and Flyout windows is that Settings is modal and doesn't vanish until the user clicks on OK or Cancel, but the Flyout vanishes as soon as it loses focus.

The Gadget HTML starts in the usual way:

```
<html>
<head>
<title>Debug</title>
```

```
<script>␣
  window.onload=function()␣
  {␣
    debug("hello debug world");␣
  }␣
```

It simply calls the debug function to display the message. The debug function, which can be added to any gadget you are trying to debug, is simple. First it sets the file that contains the HTML we want to use for the flyout and then sets the show property to true:

```
function debug(message)␣
{␣
System.Gadget.Flyout.File="debug.␣
  html";␣
System.Gadget.Flyout.show=true;␣
```

The flyout starts to load. To start using it, we set its onShow handler to a function that we are about to write:

```
System.Gadget.Flyout.onShow =␣
  showFlyout;␣
```

We retrieve its document object and save the debug message in local variables:

```
var oFlyDoc = System.Gadget.Flyout.␣
  document;␣
var text=message;␣
```

The showFlyout function is nested within the debug function. When it is triggered by the onShow event, it has access to the variables defined in the debug function. This is a sophisticated JavaScript feature called closure and it's very useful in situations such as this:

```
function showFlyout()␣
{␣
  oFlyDoc.getElementById␣
    ('flyoutContent')␣
    .innerText =text;␣
}␣
}␣
</script>␣
</head>␣
<body style="width:130;␣
  height:50">␣
</body>␣
</html>␣
```

If you try this out, you should see the flyout appear and display the appropriate message. If you don't, try it a second time. For reasons that aren't clear, the onload event sometimes isn't triggered when you reload a gadget.

COUNTDOWN TIMER GADGET

A useful gadget that isn't already provided by default is an event countdown timer, or an Are



▲ The debug flyout in action

We There Yet (AWTY) gadget. The user inputs a date, and the gadget shows how much time is left before the date. It can be used as a birthday countdown, holiday countdown or a countdown to whatever you consider important. Start a new gadget by creating a folder called AWTY.

Store within it the basic AWTY.html file:

```
<html>␣
<head>␣
  <title>AWTY</title>␣
  <script>␣
    System.Gadget.settingsUI=␣
      "settings.html";␣
  </script>␣
</head>␣
<body style="width:130; height:50␣
  " >␣
</body>␣
</html>␣
```

Also store an XML manifest called gadget.xml:

```
<?xml version="1.0" encoding="utf-8"␣
  ?>␣
<gadget>␣
  <name>AWTY</name>␣
  <version>1.0.0.0</version>␣
  <hosts>␣
    <host name="sidebar">␣
      <base type="HTML"␣
        apiVersion="1.0.0"␣
        src="AWTY.html" />␣
      <permissions>Full</␣
        permissions>␣
      <platform minPlatformVersion␣
        ="1.0" />␣
    </host>␣
  </hosts>␣
</gadget>␣
```

The first part of the gadget we have to implement is Settings.html, which is where the user can set the date of the important event. This is similar to the previous Settings file but it has three text boxes for day, month and year:

```
<html>␣
<head>␣
  <script>␣
    window.onload=function()␣
    {␣
      var Event=System.Gadget.␣
        Settings.read("Event");␣
      var d=new Date(Event);␣
      Text1.value=d.getDate();␣
      Text2.value=d.getMonth()+1;␣
      Text3.value=d.getFullYear();␣
    }␣
    System.Gadget.onSettingsClosing␣
      =␣
      function(event)␣
      {␣
        if (event.closeAction ==␣
          event.Action.commit)␣
        {␣
          d=new Date(Text3.value,␣
            Text2.value-1,Text1.␣
              value);␣
          System.Gadget.Settings.␣
            write("Event",d);␣
        }␣
        event.cancel = false;␣
      }␣
    </script>␣
```

```
</head>␣
<body style="width:250;␣
  height:250">␣
  Enter your important date␣
  <br />␣
  Day␣
  <br />␣
  <input id="Text1" type="text" />␣
  <br />␣
  Month␣
  <br />␣
  <input id="Text2" type="text" />␣
  <br />␣
  Year␣
  <br />␣
  <input id="Text3" type="text" />␣
</body>␣
</html>␣
```

The conversion from day, month and year to a Date object is via a standard constructor. It is then saved to an Event property, ready to be retrieved and unpacked into day, month and year forms when needed. You can see the unpacking process in action in the window onload event handler. The first time the Settings window is opened, the date is undefined and all the text boxes show NaN (Not a Number). You could fix this with a simple If statement.

The main gadget HTML is quite simple but with the addition of two new features: date manipulation and an auto-refresh. First we need to set up the Settings window:

```
<html>␣
<head>␣
  <title>AWTY</title>␣
  <script>␣
    System.Gadget.settingsUI=␣
      "settings.html";␣
    var Timer;␣
```

To make the gadget auto-update, we need to use the main window's Timer function. This can be set to run a specified function at a set interval, repeatedly, or once. The window onload event handler is a good place to set up the timer:

```
window.onload=function()␣
{␣
  Timer=window.setInterval(␣
    'getTime()',10000);␣
}␣
```

This specifies that the getTime() function should be called every 10,000 milliseconds (every 10 seconds). You can adjust the refresh time to suit your particular application. Although not used in this gadget, the setInterval function returns a code that identifies the timer that has been created. This code can be used by other functions to start, stop and change the time interval, so it's worth storing in a global variable. In this case, it's stored in Timer.

TIME OUT

The getTime function does the actual work of retrieving the target date, retrieving the current date, finding the difference between the two and then displaying a reasonably formatted display of the result.

Retrieving the target date is easy:

```
getTime=function()␣
{␣
```


SAMS

Silverlight 1.0
Unleashed

★★★★

£20

From www.amazon.co.uk

Microsoft's Silverlight browser plug-in is a competitor to Adobe's Flash and Sun's JavaFX, and is designed to allow developers to create beautiful-looking websites. This book is a guide to the new technology. Author Adam Nathan was the founding developer of Popfly, Microsoft's first product built with Silverlight, so he's in a good position to introduce the core concepts and techniques for using the technology.

The book uses a lot of code snippets to help you learn. They're colour-coded, so they're easy to follow, and it helps when you get around to writing your own projects. The layout also contains boxouts with Tips, Warning and Digging Deeper sections that expand on key points.

The information given is very good. If we have one complaint, it's that the book is rather thin. It is a great introduction to the topic, but those who want to take things further will be disappointed by the lack of detail, particularly for XAML and JavaScript.

This aside, however, *Silverlight 1.0 Unleashed* is a very good introduction to a new technology.

SUMMARY

- ✓ Good introduction to subject
- ✓ Code easy to read
- ✗ Not much detail

ISBN 0672330075
PAGES 258

O'REILLY

C# 3.0 in a Nutshell
(3rd Edition)

★★★★

£22

From www.amazon.co.uk

Although it is part of the Nutshell series, this is a seriously thick book. While it's the third edition about C#, it's the first to include information on C# 3.0. For readers familiar with C# 2.0, the book includes over a hundred pages on the new features introduced into the language in Visual Studio 2008, including Language Integrated Query (LINQ).

You don't need prior acquaintance with C# to benefit from this book, as it covers all aspects of the language. It is equally useful if you're working in C# 2.0, since sections that apply only to C# 3.0 are clearly marked.

However, readers need to have some general programming experience. The first three chapters relate to C#, starting with syntax, types and variables and progressing to advanced topics such as unsafe code and pre-processor directives. The remaining chapters cover the core .NET Framework, which includes collections, LINQ, XML, streams, networking, reflection, security, threading, application domains, working with native DLLs and diagnostics.

C# programmers are sure to find this a useful addition to their bookshelves.

SUMMARY

- ✓ Covers the new technology
- ✓ A great reference for C#
- ✗ Daunting for beginners

ISBN 0596527578
PAGES 838

```
Event=System.Gadget.␣
Settings.read("Event");␣
```

We want to do work only if the target has been defined by the user:

```
if(Event!="")␣
{␣
```

If the Event is defined then it contains a date and time formatted string that can be used to create a JavaScript date object:

```
var d=new Date(Event);␣
```

Later on we are also going to need a string formatted as a simple date (dd/mm/yyyy) and this can be constructed using the date object:

```
target=d.getDate()+"/"␣
+(d.getMonth()+1) + "/"␣
+d.getFullYear();␣
```

To get the current date, just get the default date object constructor:

```
var Now=new Date();␣
```

The getTime method returns the number of milliseconds since midnight on 1st January 1970. Most date arithmetic in JavaScript is easier to do by working in milliseconds and then converting to whatever time/date units we really want to work with. In this case, we simply subtract the two dates in milliseconds since the start of 1970:

```
var diff=d.getTime()-Now.getTime();␣
```

To convert this to number of days, we divide by 1,000 to give seconds, then by 60 to give minutes, followed by 60 to give hours, and finally by 24 to give days:

```
diff=diff/1000/60/60/24;␣
```

This gives a result that is in days and fractional days. If you want to convert to days and hours, say, then you would need to truncate the result to an integer and convert the fractional day into hours, minutes and seconds. This isn't difficult, but it's potentially confusing. In this example, we simply round the result to six significant digits:

```
diff=Math.round(diff*1000000)/␣
1000000;␣
```

property allows us to enter any HTML we care to between the tags:

```
System.Gadget.document.␣
getElementById('Display').␣
innerHTML=target+" <br/>␣
"+diff;␣
}␣
</script>␣
</head>␣
```

Here, the target date and the number of days to the target are separated by a line break. You can include additional HTML formatting tags such as or <h1> to make the text stand out.

All that remains is the body of the page:

```
<body style="width:130; height:150">␣
Number of days until␣
<br/>␣
<div id="Display">␣
</div>␣
</body>␣
</html>␣
```

You can see the pair of <div> tags with the id Display, defining where the date information will appear on the page, along with some additional text and formatting.

As long as you have stored everything in the correct directory and given everything the correct names, you should now be able to load your gadget, set a target date and watch as the countdown proceeds. There is lots of scope for making the user interface look more exciting. A background picture, an icon and some colour would make an immediate impression.

You can use the debug window function to find out what is happening within timer interrupts, but you have to be careful, because the debug window will be re-created each time the event handler is called, and this can be very confusing. **CS**



▲ The date Settings window

DISPLAYING THE RESULT

We are now ready to display the result. We could use a text box or some other display control, but we can also enter text directly into the HTML of the webpage. The easiest way to do this is to use a <div> or a pair of tags complete with an id that can be retrieved using the DOM getElementById function.

If you are going to get very far with gadgets or JavaScript/HTML in general, you will have to master the use of the DOM. In this case, we assume that there is a tag with id="Display". The getElementById method returns the DOM object corresponding to the tags and the .innerHTML